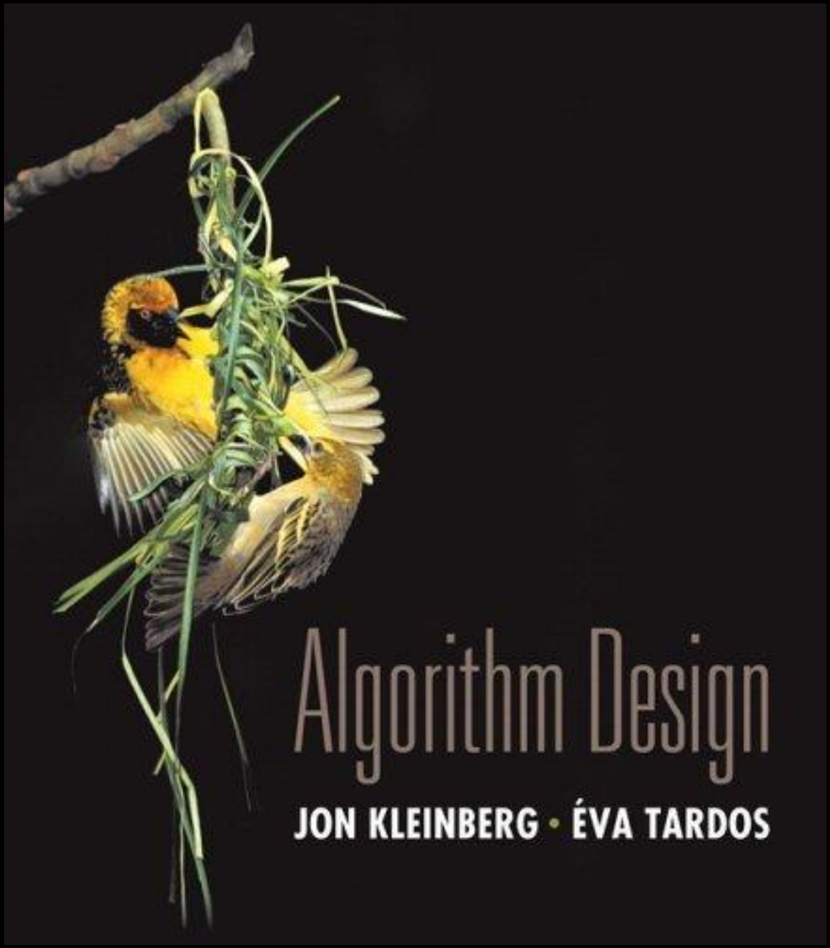
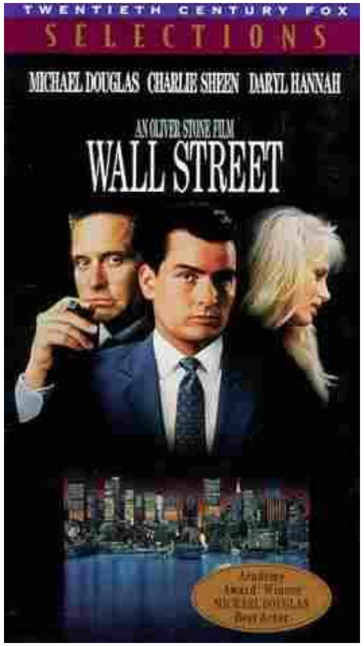


Chapter 4

Greedy Algorithms

Greed is good. Greed is right. Greed works.
Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.

- Gordon Gecko (Michael Douglas)

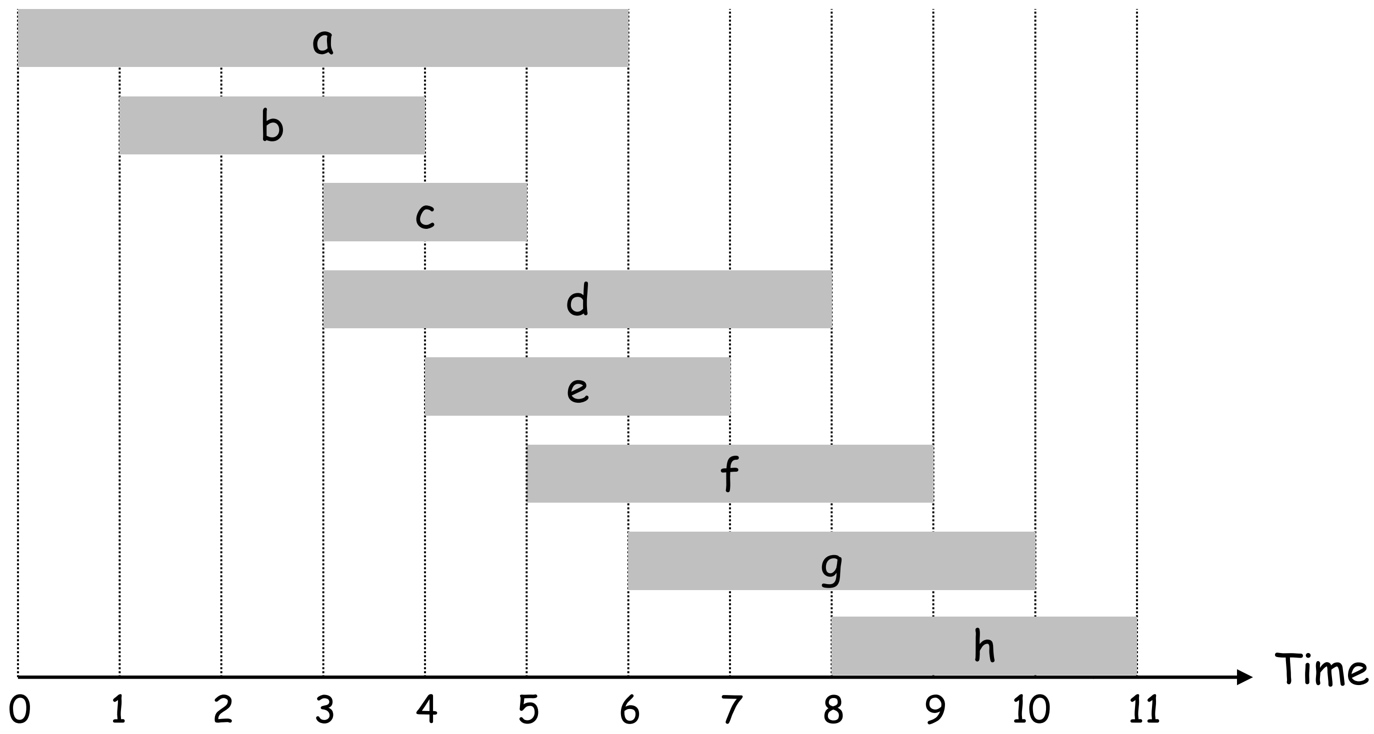


4.1 Interval Scheduling

Interval Scheduling

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

- [Earliest start time] Consider jobs in ascending order of start time s_j.
- [Earliest finish time] Consider jobs in ascending order of finish time f_j.
- [Shortest interval] Consider jobs in ascending order of interval length $f_j - s_j$.
- [Fewest conflicts] For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

breaks earliest start time

breaks shortest interval

breaks fewest conflicts

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

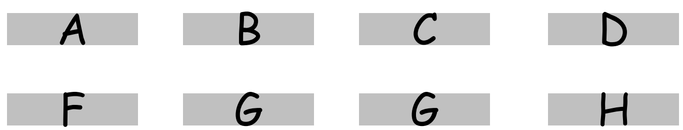
```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

  jobs selected
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation. $O(n \log n)$.

- Remember job j^* that was added last to A.
- Job j is compatible with A if $s_j \geq f_{j^*}$.

Interval Scheduling: Analysis

Optimal Solutions. There may be more than one optimal solutions for the same instance



2x2x2x2 = 16 optimal solutions

Prefix of a solution. Define the prefix of length I of a solution S as the first I jobs of S

The prefix of length 2 of AGD is AG

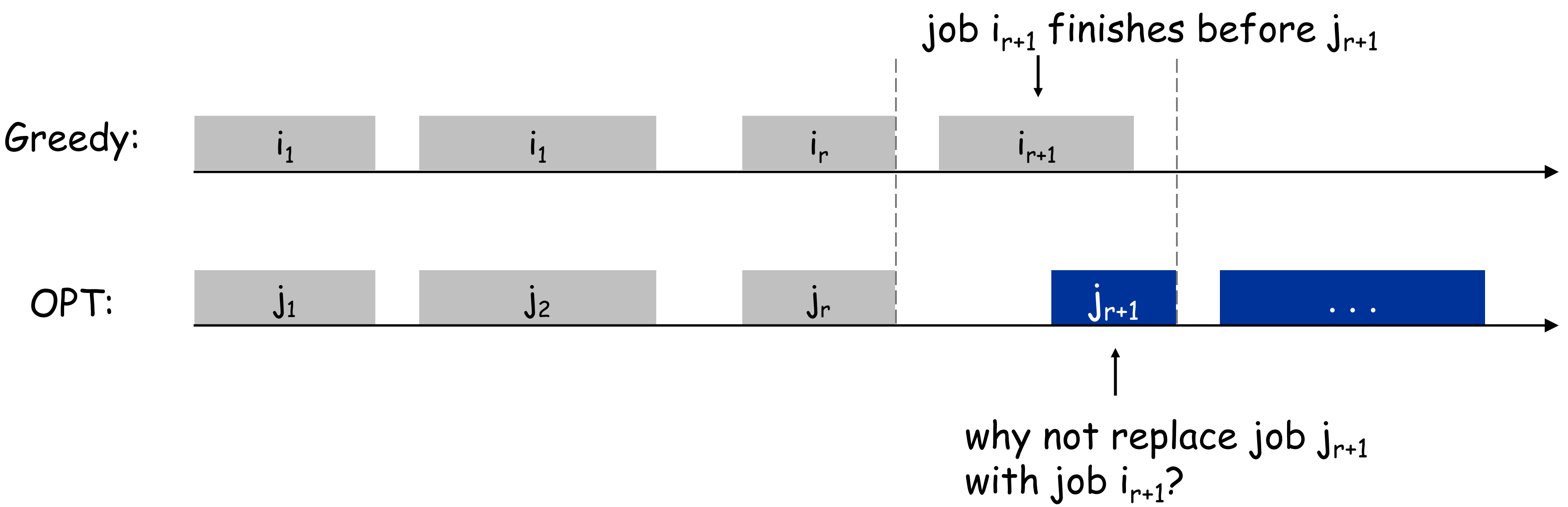
Similarity of two solutions S e S'. The largest j such that S and S' shares a prefix of length j

Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution which is more similar to greedy's solution

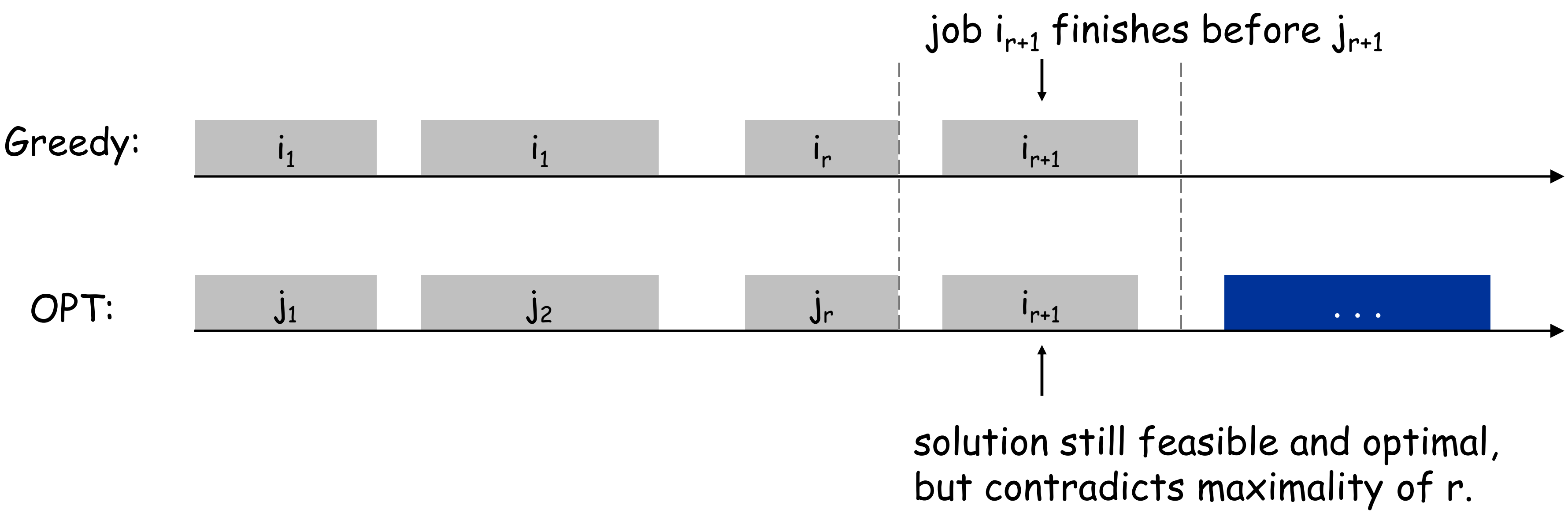


Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution which is more similar to greedy's solution



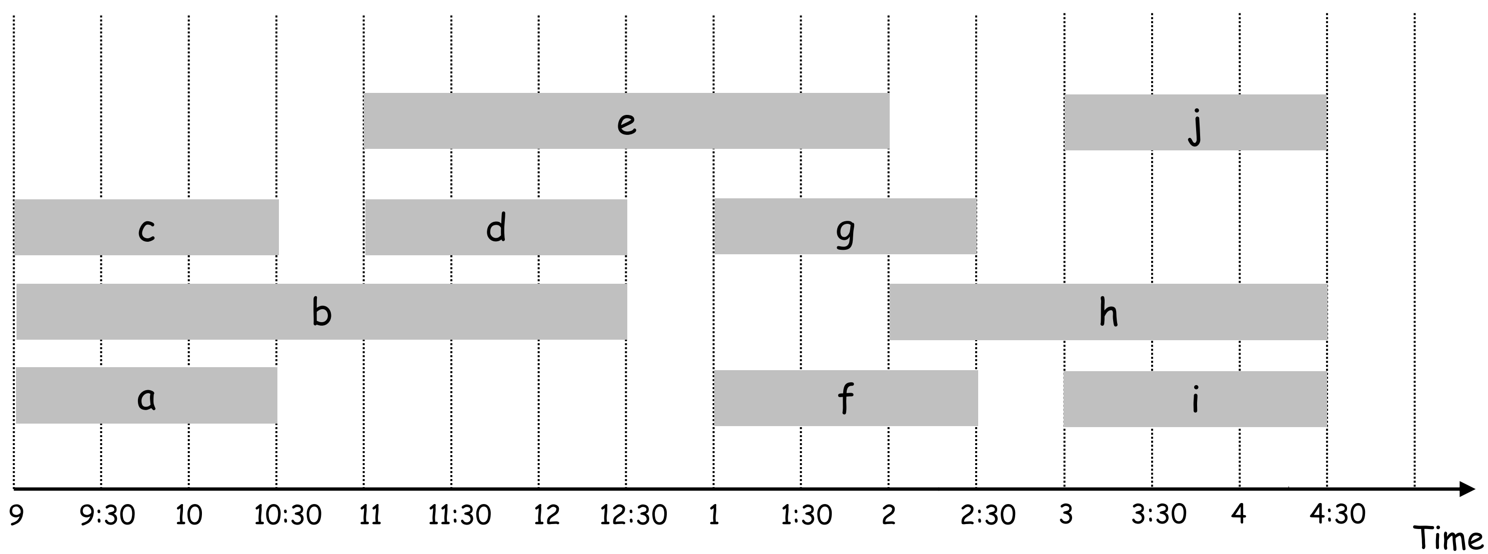
4.1 Interval Partitioning

Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

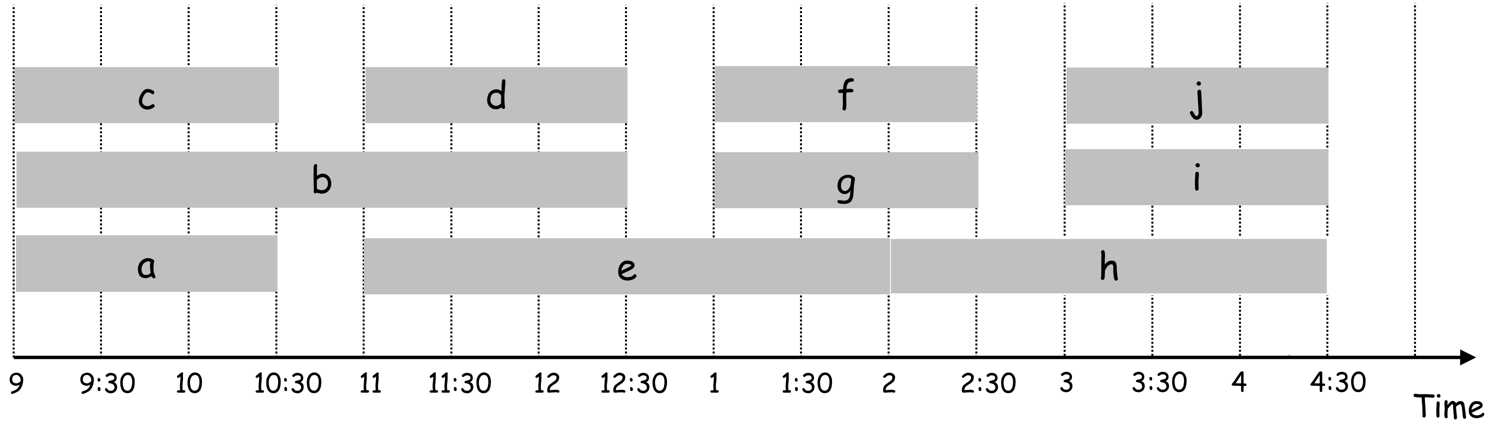
Ex: This schedule uses 4 classrooms to schedule 10 lectures.



Interval Partitioning

Interval partitioning.
- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses only 3.



Interval Partitioning: Lower Bound on Optimal Solution

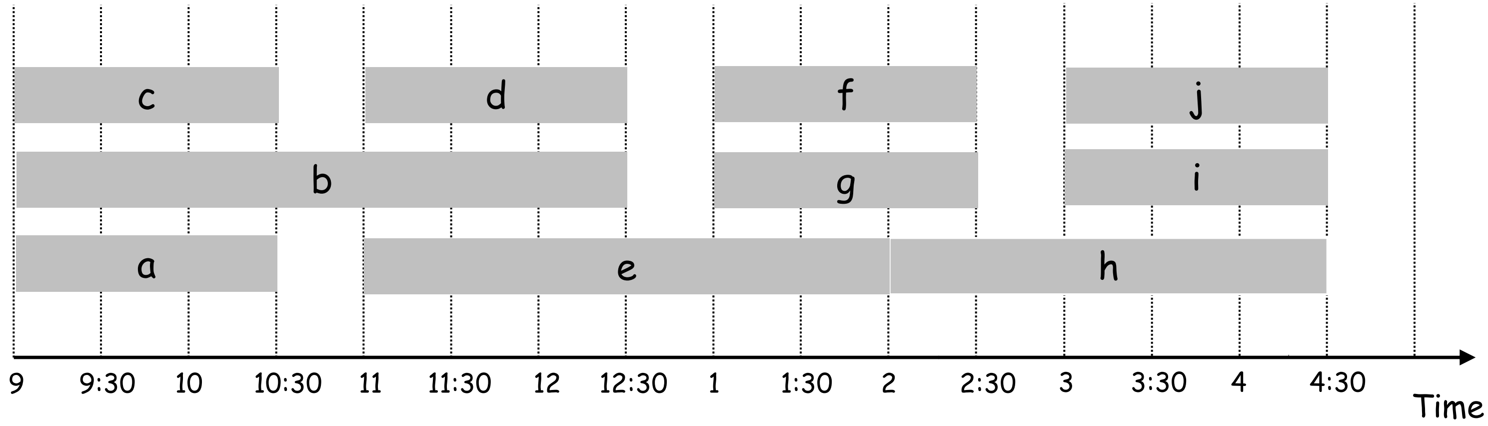
Def. The depth of a set of open intervals is the maximum number that contain any given time.

Key observation. Number of classrooms needed $\geq$ depth.

Ex: Depth of schedule below = 3 $\Rightarrow$ schedule below is optimal.

a, b, c all contain 9:30

Q. Does there always exist a schedule equal to depth of intervals?



Interval Partitioning: Greedy Algorithm

Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0   ← number of allocated classrooms

for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

Interval Partitioning: Greedy Algorithm

Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0   ← number of allocated classrooms

for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

Interval Partitioning: Greedy Analysis

Observation. Greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Greedy algorithm is optimal.
Pf.

- Let d = number of classrooms that the greedy algorithm allocates.
- Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all d-1 other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
- Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
- Key observation $\Rightarrow$ all schedules use $\geq$ d classrooms. ▪

Interval Partitioning: Greedy Algorithm

Implementation. $O(n^2)$.

- For each classroom k, maintain the finish time of the last job added.
- Keep the classrooms in a list

```
if (lecture j is compatible with some classroom k)
   Compare start time of j with the finish time of each
   class: o(n) time

schedule lecture j in classroom k
   Replace the current finish time of of class k to f_j:
   O(1) time

allocate a new classroom d + 1
schedule lecture j in classroom d + 1
   Insert a new class in the list of classes with finish
   time f_j: o(1) time
```

Interval Partitioning: Greedy Algorithm

Implementation. O(n log n).

- For each classroom k, maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

```
if (lecture j is compatible with some classroom k)
   Compare start time of j with the finish time of the
   class at the root of the heap

schedule lecture j in classroom k
   Change the priority of the root of heap to f_j
   Updated the heap : O(log n ) time

allocate a new classroom d + 1
schedule lecture j in classroom d + 1
   Insert a new class at the heap with priority f_j:
   o(log n)
```

4.2 Scheduling to Minimize Lateness

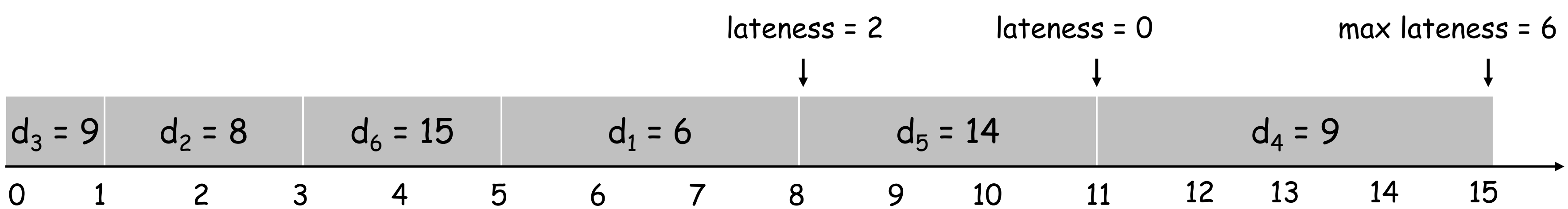
Scheduling to Minimizing Lateness

Minimizing lateness problem.

- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max \{ 0, \; f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

- [Earliest deadline first] Consider jobs in ascending order of deadline d_j.

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

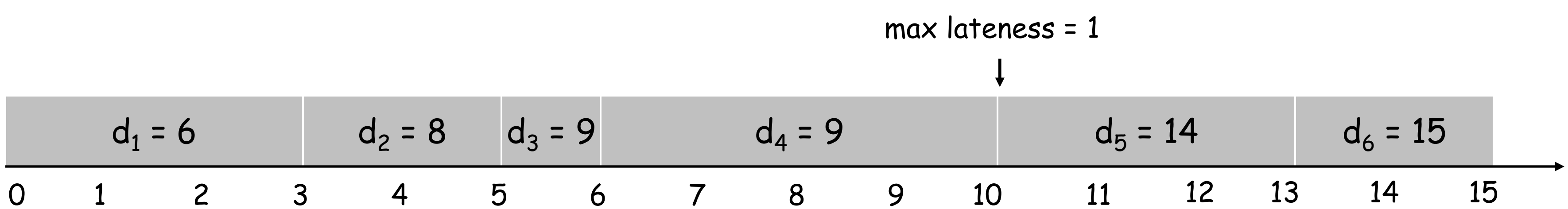
counterexample

Minimizing Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

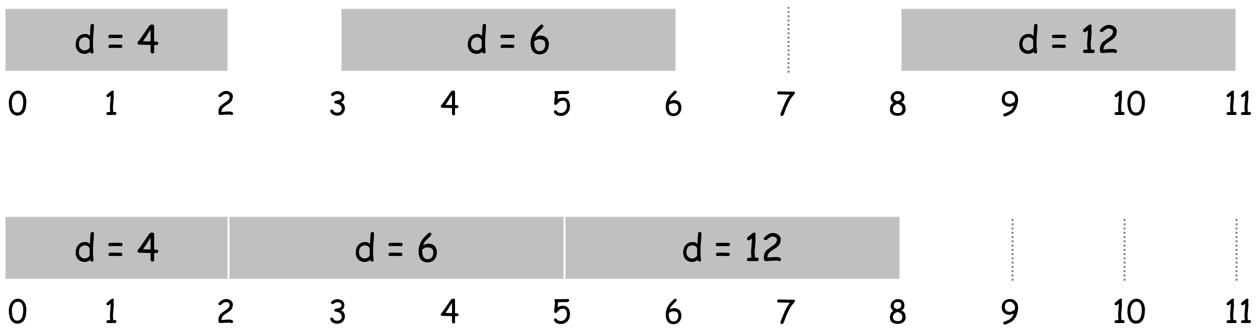
Sort n jobs by deadline so that $d_1 \le d_2 \le \ldots \le d_n$

$t \leftarrow 0$
for j = 1 to n
  Assign job j to interval $[t, t + t_j]$
  $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
  $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$



Minimizing Lateness: No Idle Time

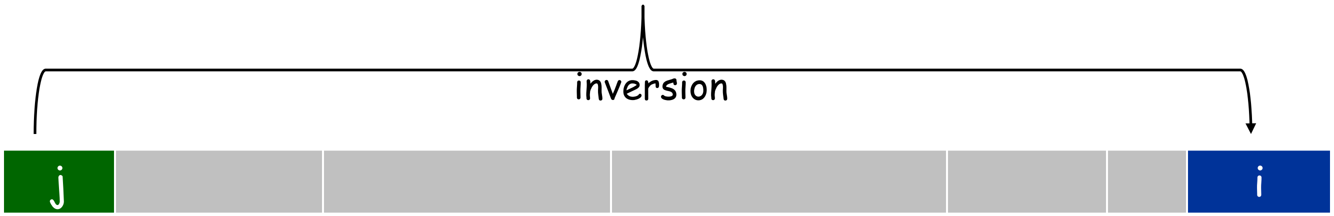
Observation. There exists an optimal schedule with no idle time.



Observation. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.

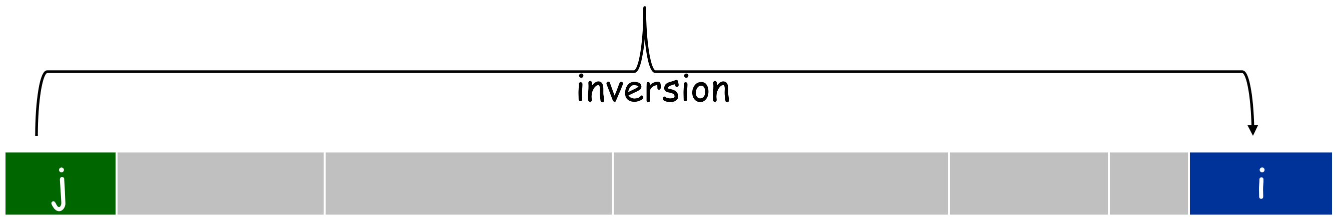


Observation 1. Greedy schedule has no inversions. All schedules with no inversions have the same lateness

- In a schedule with no inversions, the lateness of a job j with deadline d is the given by max{0, f – d}, where f is the completion time of the last scheduled job of deadline j.

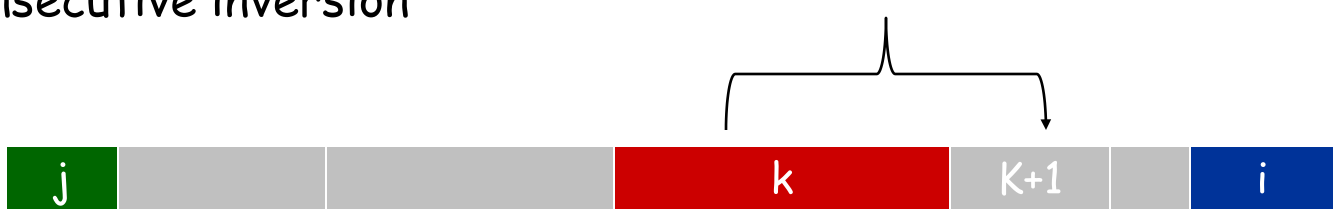
Minimizing Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.



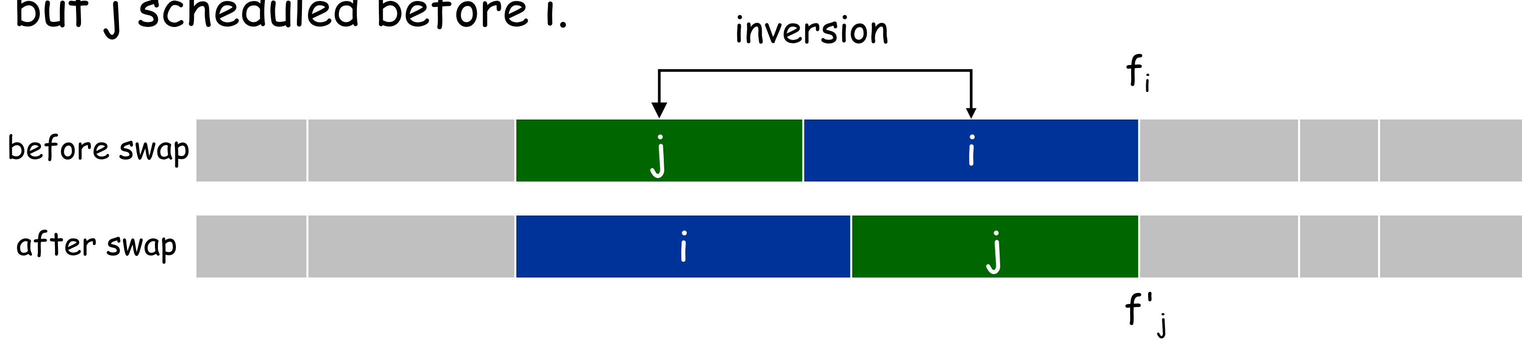
Observation 2. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

To see that consider the job k with largest deadline among the jobs between j and i. In case of ties, pick the rightmost one. k and k+1 form a consecutive inversion



Minimizing Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.



Claim. Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Pf. Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job j is late:

$$
\begin{aligned}
\ell'_j &= f'_j - d_j && \text{(definition)} \\
&= f_i - d_j && \text{(} j \text{ finishes at time } f_i \text{)} \\
&\leq f_i - d_i && (i < j) \\
&\leq \ell_i && \text{(definition)}
\end{aligned}
$$

Minimizing Lateness: Analysis of Greedy Algorithm

Theorem. Greedy schedule S is optimal.

Pf. Define S* to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

- Can assume S* has no idle time.
- If S* has no inversions, then S = S*.
- If S* has an inversion, let i-j be an adjacent inversion.
  - swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
  - this contradicts definition of S* ▪

Greedy Analysis Strategies

Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

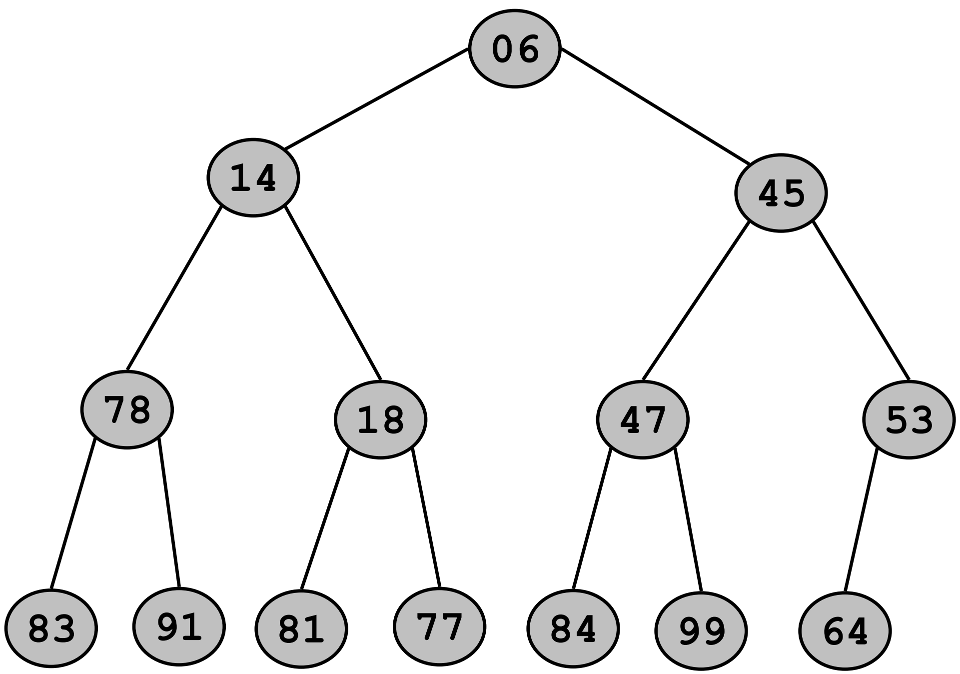
Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Binary Heap: Definition

Binary heap.

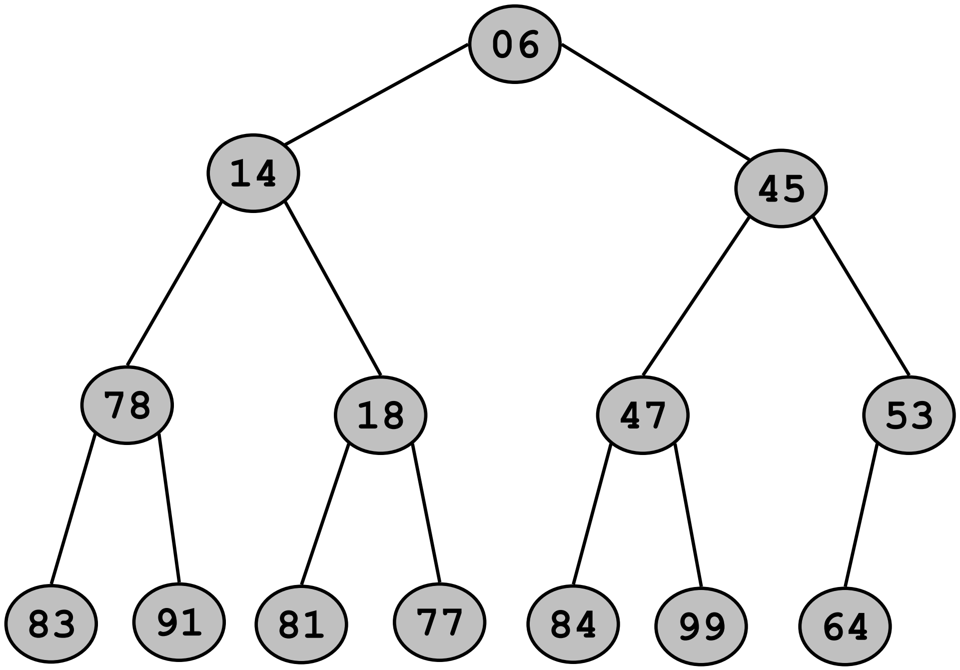
- Almost complete binary tree.
  - filled on all levels, except last, where filled from left to right
- Min-heap ordered.
  - every child greater than (or equal to) parent



Binary Heap: Properties

Properties.

- Min element is in root.
- Heap with N elements has height = $\lfloor \log_2 N \rfloor$.

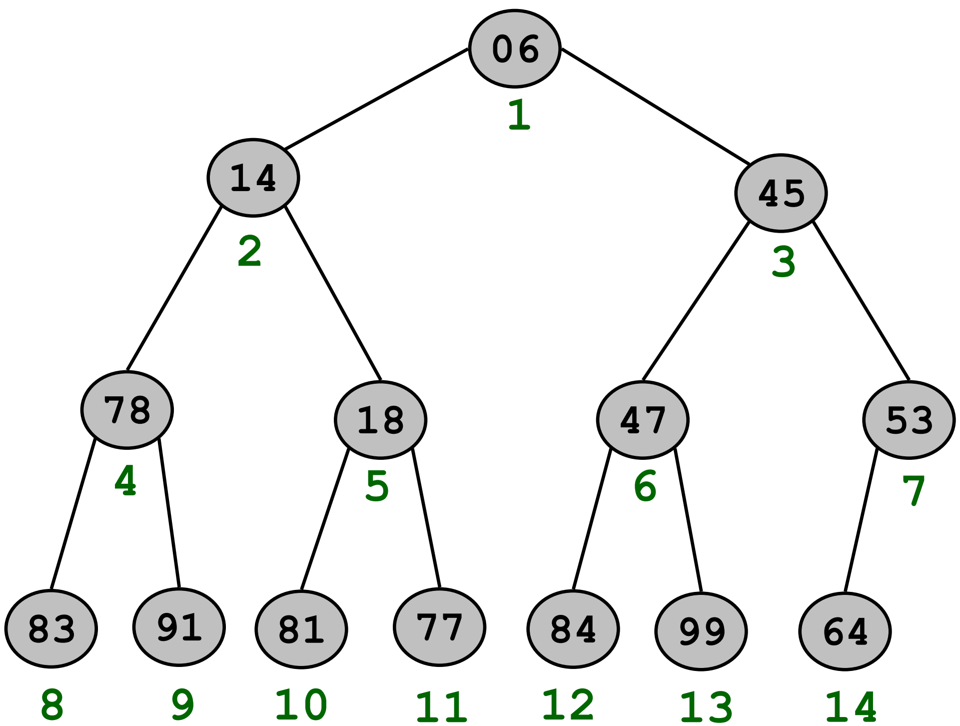


N = 14
Height = 3

Binary Heaps: Array Implementation

Implementing binary heaps.

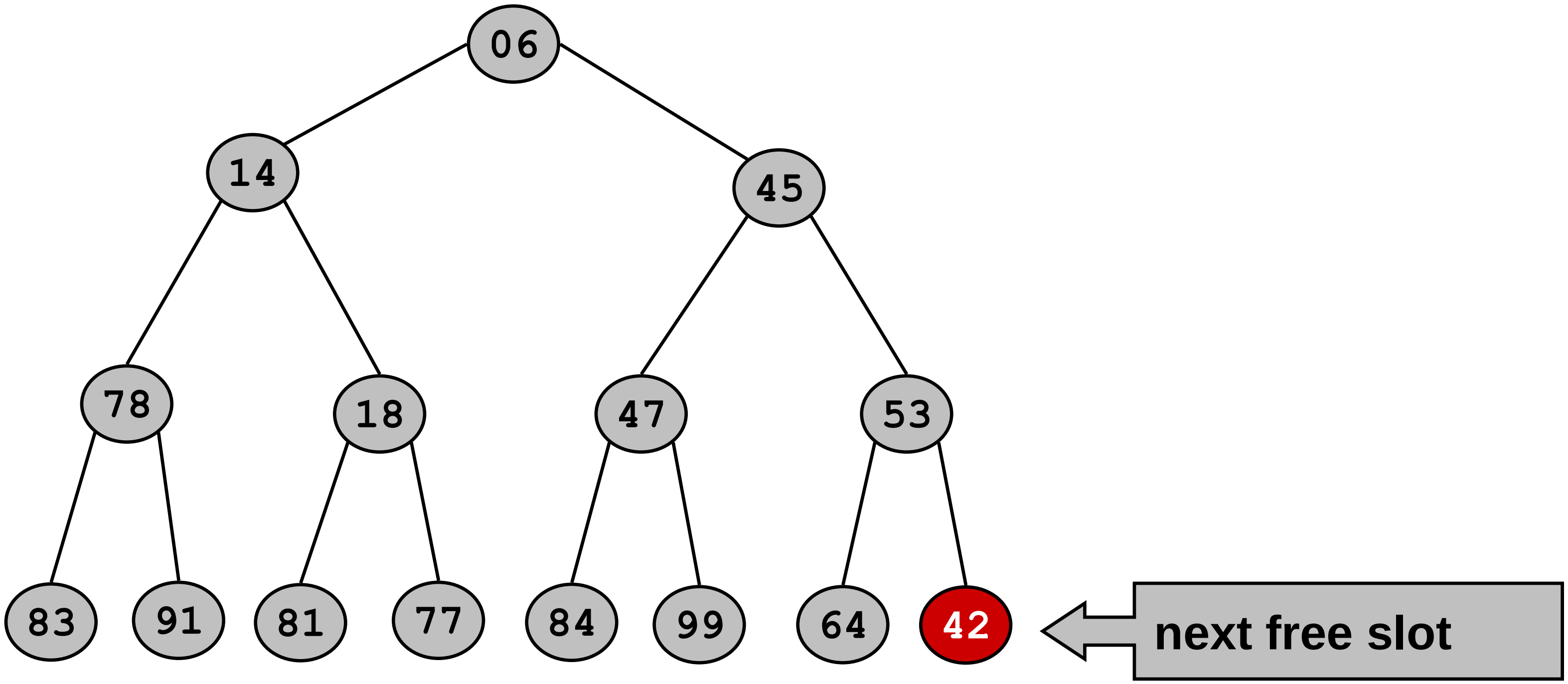
- Use an array: no need for explicit parent or child pointers.
  - `Parent(i)  =` $\lfloor i/2 \rfloor$
  - `Left(i)    = 2i`
  - `Right(i)   = 2i + 1`



Binary Heap: Insertion

Insert element x into heap.

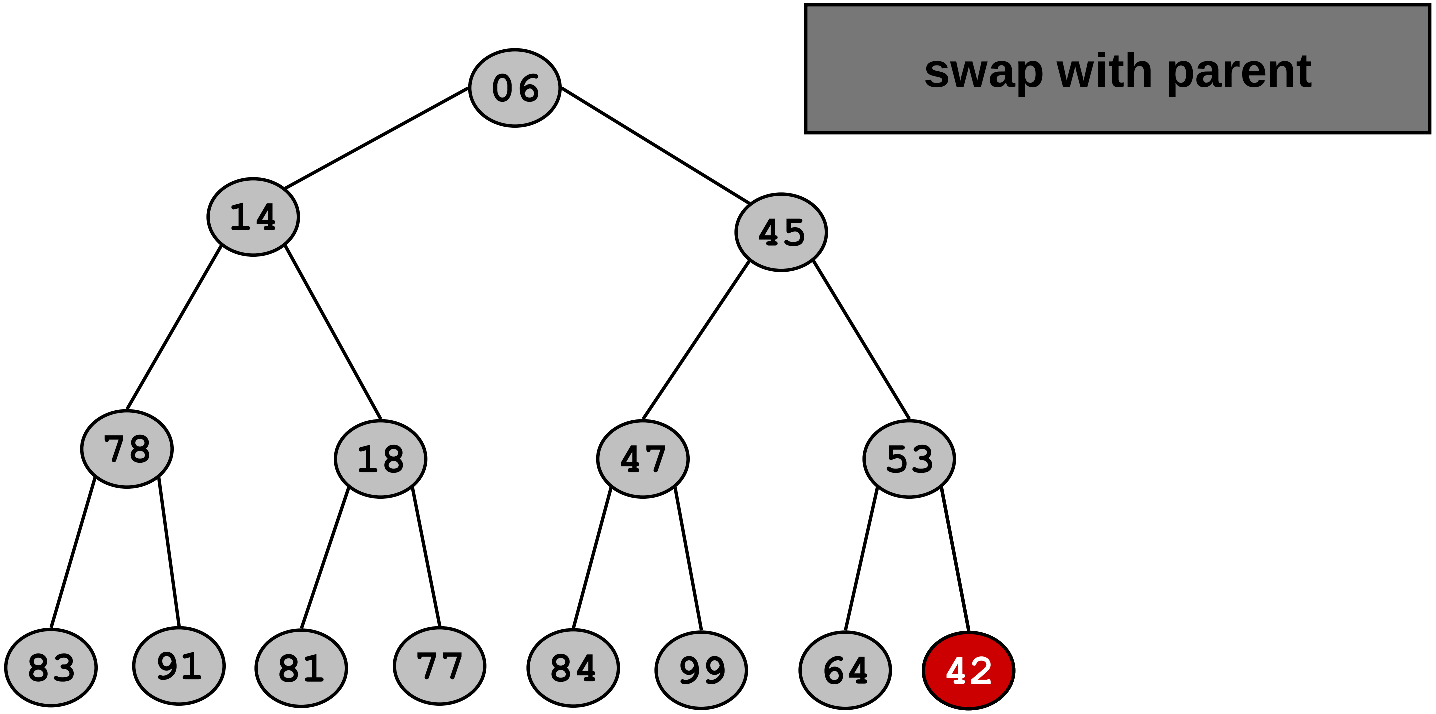
- Insert into next available slot.
- Bubble up until it's heap ordered.
  - Peter principle: nodes rise to level of incompetence



Binary Heap: Insertion

Insert element x into heap.

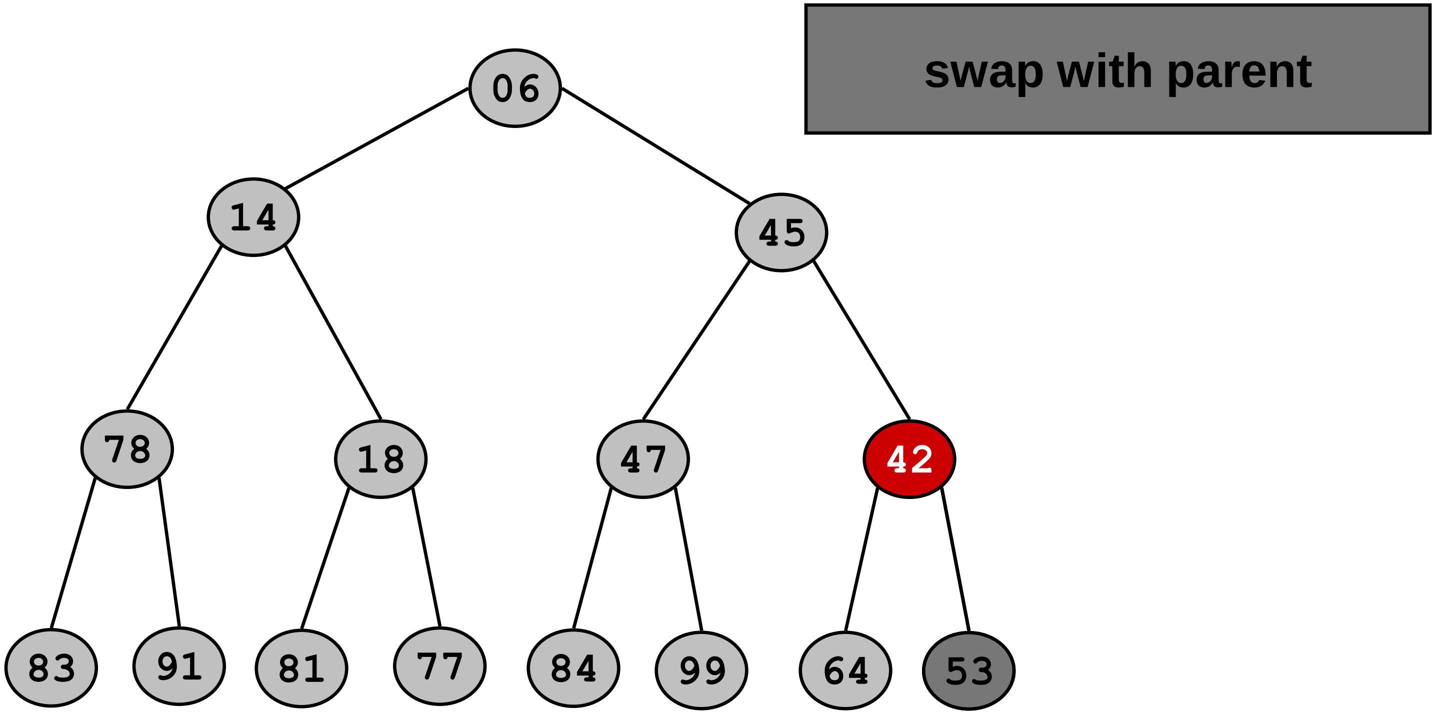
- Insert into next available slot.
- Bubble up until it's heap ordered.
  - Peter principle: nodes rise to level of incompetence



Binary Heap: Insertion

Insert element x into heap.

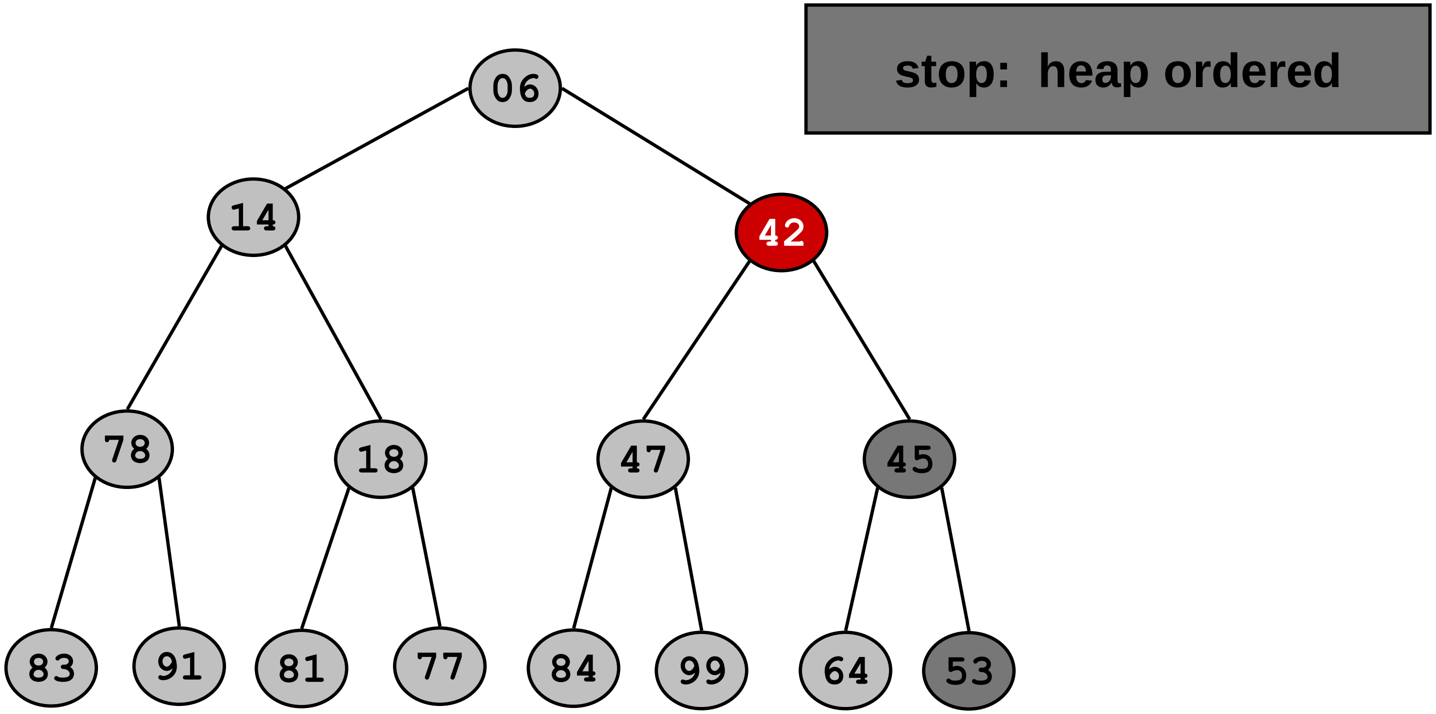
- Insert into next available slot.
- Bubble up until it's heap ordered.
  - Peter principle: nodes rise to level of incompetence



Binary Heap: Insertion

Insert element x into heap.

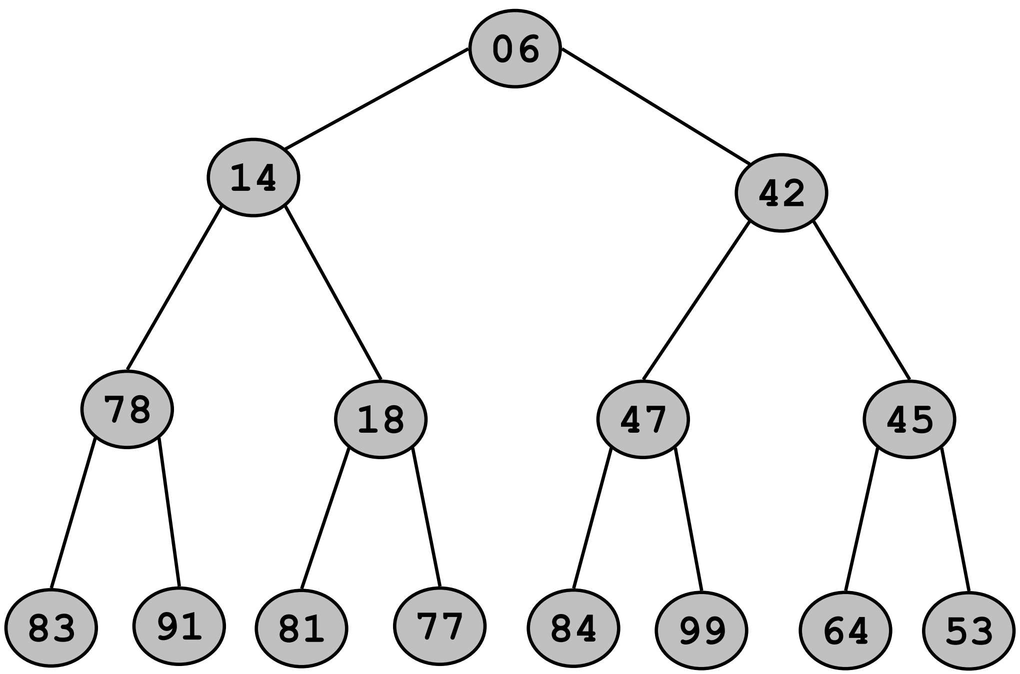
- Insert into next available slot.
- Bubble up until it's heap ordered.
  - Peter principle: nodes rise to level of incompetence
- O(log N) operations.



Binary Heap: Decrease Key

Decrease key of element x to k.

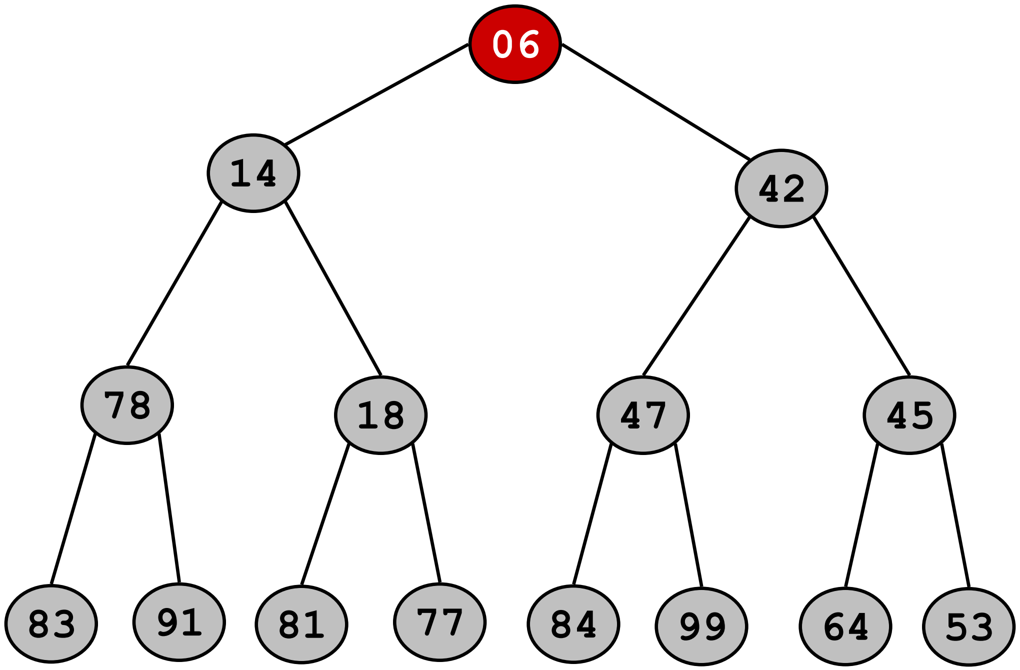
- Bubble up until it's heap ordered.
- O(log N) operations.



Binary Heap: Delete Min

Delete minimum element from heap.

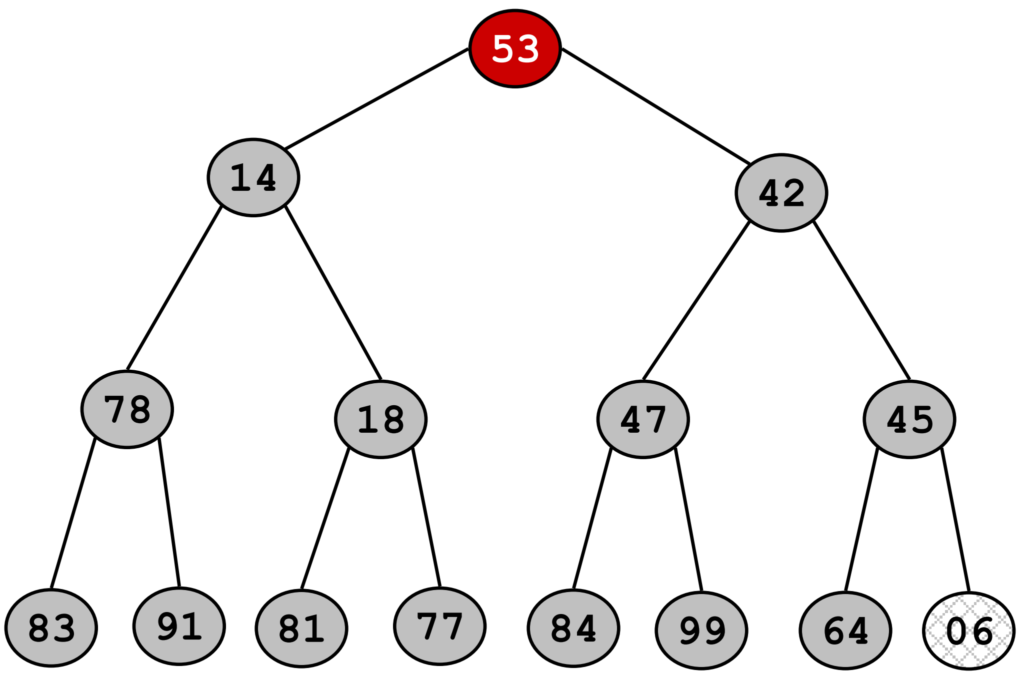
- Exchange root with rightmost leaf.
- Bubble root down until it's heap ordered.
  - power struggle principle: better subordinate is promoted



Binary Heap: Delete Min

Delete minimum element from heap.

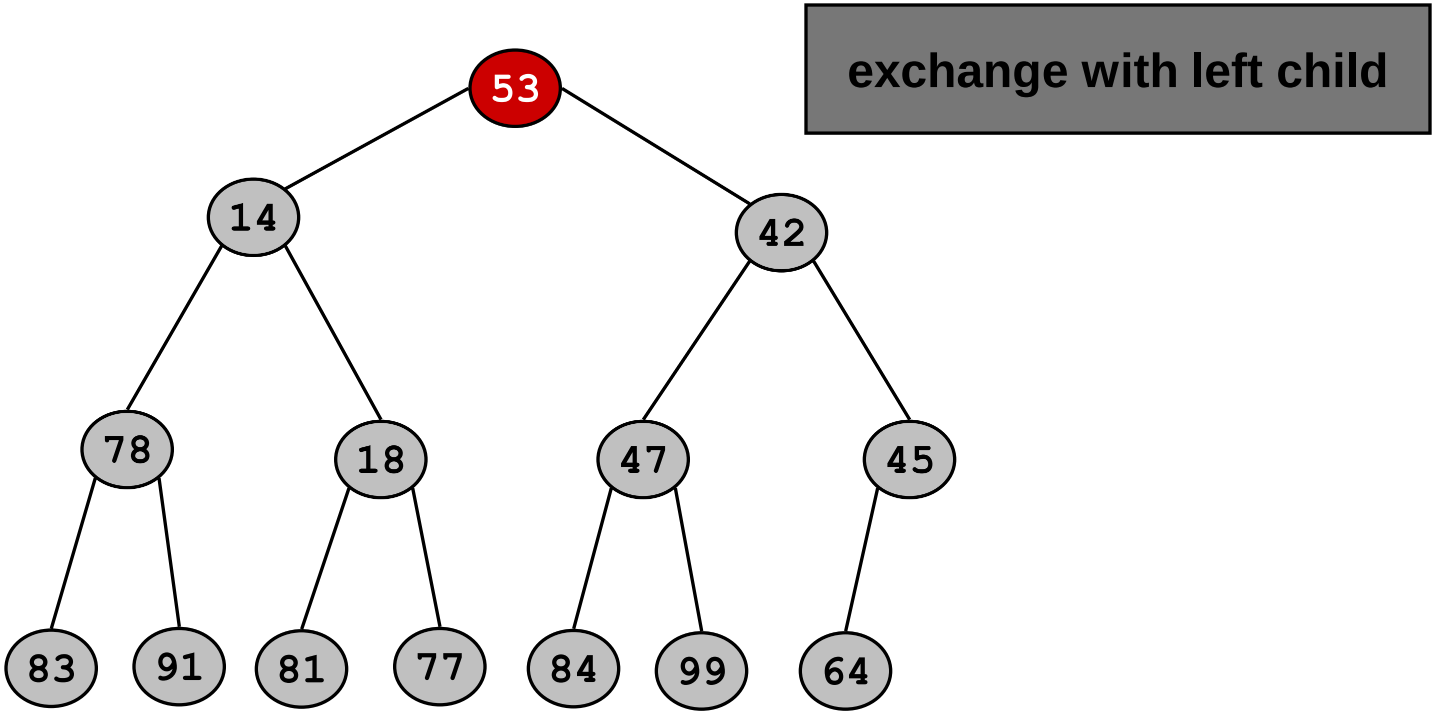
- Exchange root with rightmost leaf.
- Bubble root down until it's heap ordered.
  - power struggle principle: better subordinate is promoted



Binary Heap: Delete Min

Delete minimum element from heap.

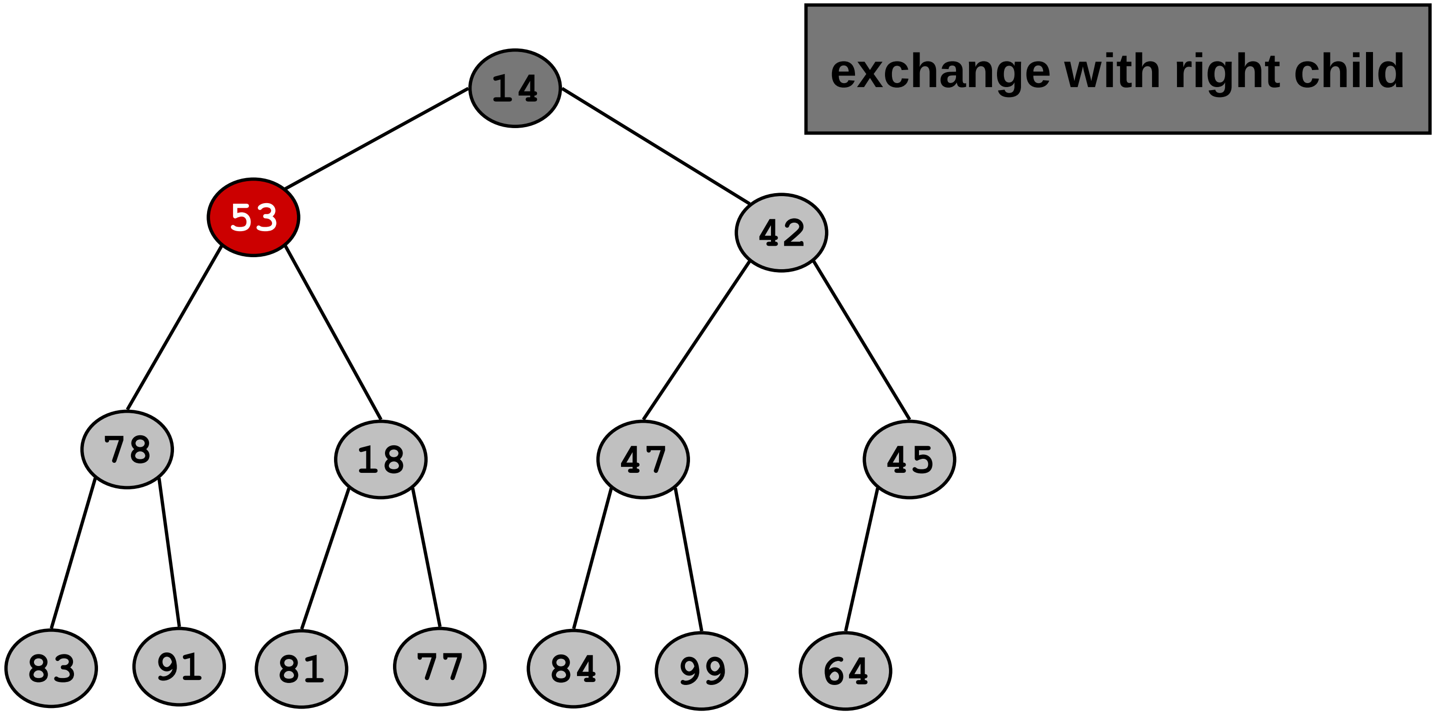
- Exchange root with rightmost leaf.
- Bubble root down until it's heap ordered.
  - power struggle principle: better subordinate is promoted



Binary Heap: Delete Min

Delete minimum element from heap.

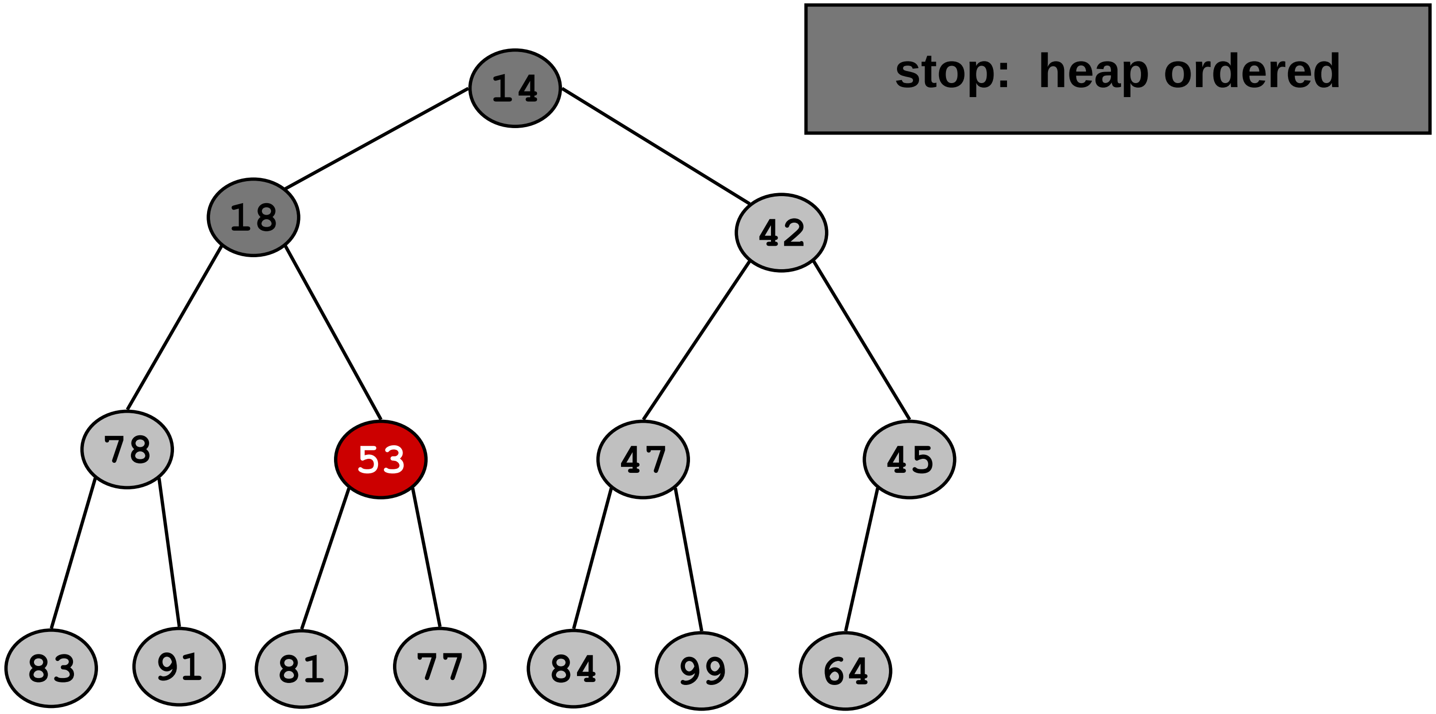
- Exchange root with rightmost leaf.
- Bubble root down until it's heap ordered.
  - power struggle principle: better subordinate is promoted



Binary Heap: Delete Min

Delete minimum element from heap.

- Exchange root with rightmost leaf.
- Bubble root down until it's heap ordered.
  - power struggle principle: better subordinate is promoted
- O(log N) operations.



Binary Heap: Heapsort

Heapsort.

- Insert N items into binary heap.
- Perform N delete-min operations.
- O(N log N) sort.
- No extra storage.

Exercices

- Simule a execução do heapsort para lista dada pelas prioridades
  18 25 41 34 10 52 50 48

Exercices

- A operação descer(i), desce com o i-ésimo elemento até que ele esteja na posição correta, ou seja, tenha prioridade menor que seus filhos
- A operação subir(i), sobe com o i-ésimo elemento até que ele esteja na posição correta, ou seja, tenha prioridade maior que seu pai

- Dado uma lista a[1],…,a[n], de n prioridades, considere os seguintes métodos para construir um heap

```
HeapBuild1
For i=n/2 downto 1
        Descer(a[i])
```

```
HeapBuild2
For i=2 to n
    Subir (a[i])
```

Análise a complexidade dos dois algoritmos

Solution

HeapBuild 2

$$\sum_{i=2}^{n} \lfloor \log i \rfloor = \Theta(\log n)$$

HeapBuild 1

$$\sum_{i=i}^{\log n-1} 2^{i} i = \Theta(n)$$

Exercices

- Um sistema consiste de uma série de componentes $C=\{o(1),\ldots,o(n)\}$.
- O sistema é dito confiável se e somente se **todos** os seus componentes funcionam.
- O custo para testar o funcionamento de $o(i)$ é $c(i)$
- Probabilidade de $o(i)$ falhar é $p(i)$.
- Assuma que as probabilidades de falha são mutuamente independentes
- Deseja-se projetar um algoritmo para testar a confiabilidade do sistema minimizando o custo esperado dos testes.

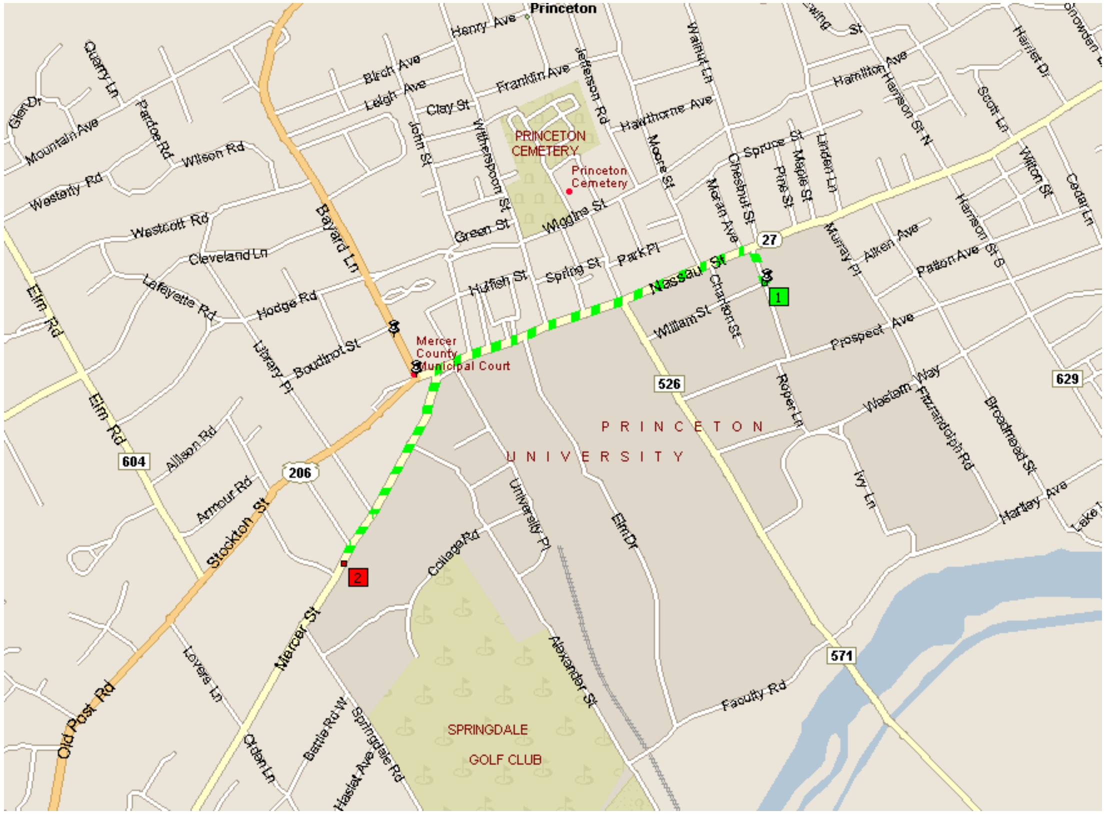
A) Considere um algoritmo que testa os componentes em ordem crescente de custos. Ele é ótimo ? Por que ?

B) Considere agora um algoritmo que testa os componentes em ordem crescente de $c(i) / p(i)$. Ele é ótimo ? Por que ?

Solution

- Considere uma ordem P para os componentes. Seja p_i a probabilidade de falha do i-ésimo componente da lista e c_i seu custo. Então
  - COntribuição do j-ésimo elemento componente da lista para o custo esperado é $(1-p1)(1-p_2)\ldots(1-p_{j-1})c_j$.
  - O custo esperado da ordem é a soma das contribuições de seus componentes
  - Mostre que em uma solução com inversões é possível trocar a ordem de dois componentes consecutivos melhorando o custo

4.4 Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house

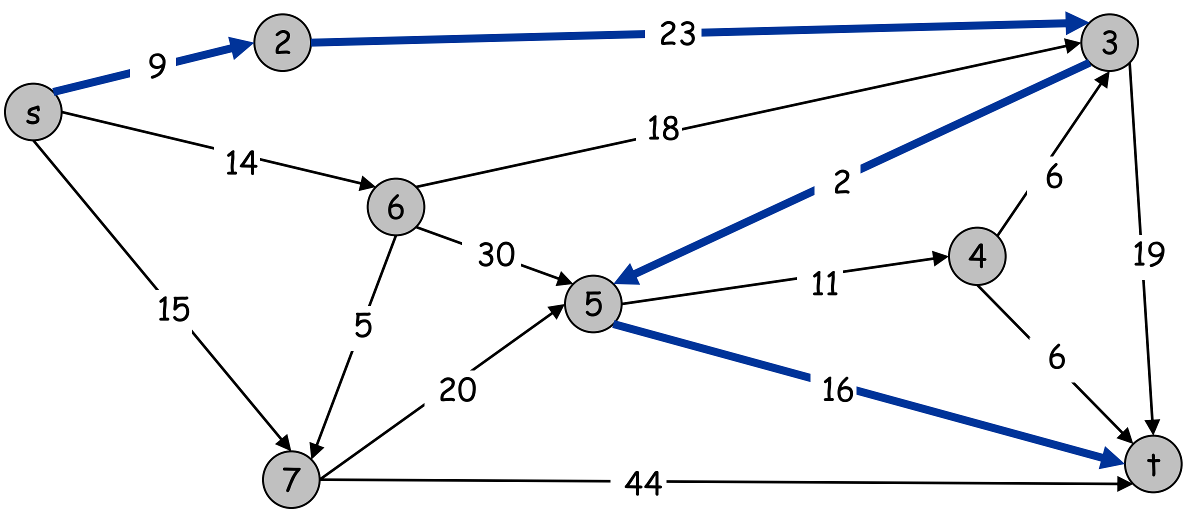
Shortest Path Problem

Shortest path network.

- Directed graph $G = (V, E)$.
- Source s, destination t.
- Length c_e = length of edge e. (non-negative numbers)

Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path

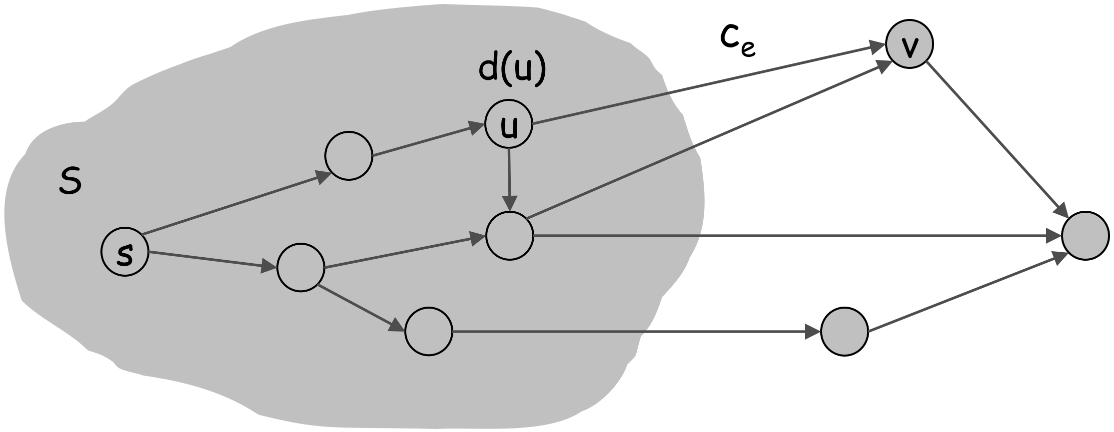


Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 50.

Dijkstra's Algorithm

Approach

- Find the node closest to the s, then the second closest, then the third closest, and so on …
- Key observation:
  - the shortest path from s to the k-th closest node can be decomposed as the shortest path from s to the i-th closest node (for some i<k) and an edge from the i-th closest node to the k-th closest node.



Dijkstra's Algorithm

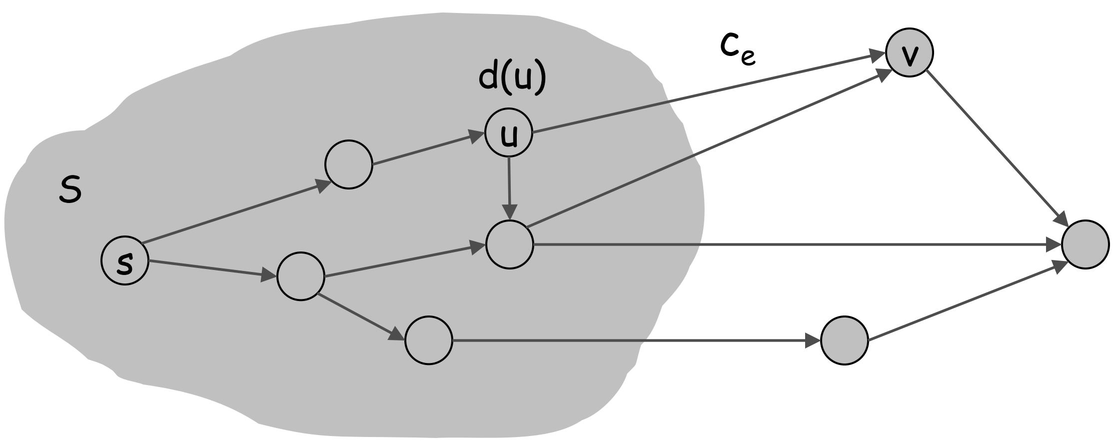
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + c_e ,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



Dijkstra's Algorithm

Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + c_e ,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).

Complexity (Naïve Implementation)

- n loops, one for each node
- (n+m) to find the the node with minimum π
- ➔ O(n(n+m)) time

Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain $\pi(v) = \min_{e=(u,v):u \in S} d(u) + c_e$.

- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update

$$\pi(w) = \min\ \{\ \pi(w),\ \pi(v) + c_e\ \}.$$

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.'

PQ Operation	Dijkstra	Array	Binary heap
Insert	n	n	log n
ExtractMin	n	n	log n
ChangeKey	m	1	log n
IsEmpty	n	1	1
Total		n^2	m log n

† Individual ops are amortized bounds

Dijkstra's Algorithm

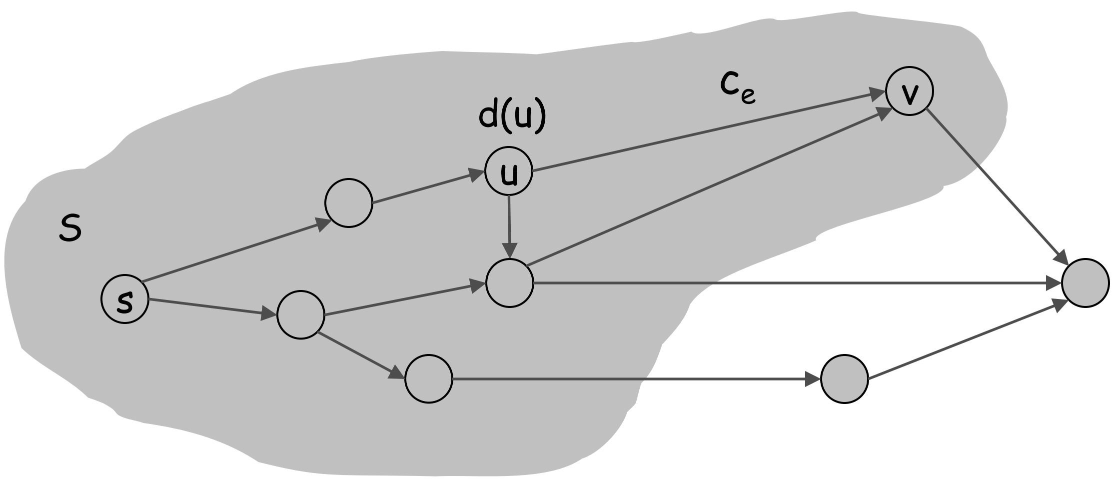
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + c_e ,$$

add v to S, and set d(v) = π(v).

shortest path to some u in explored part, followed by a single edge (u, v)



Dijkstra's Algorithm: Proof of Correctness

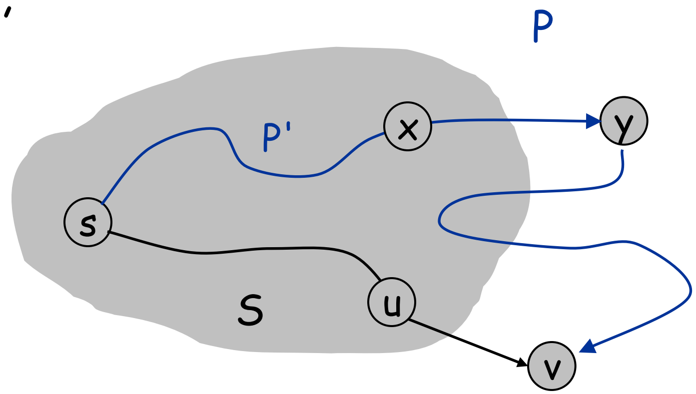
Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.

Pf. (by induction on $|S|$)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S, and let u-v be the chosen edge.
- The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
- Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



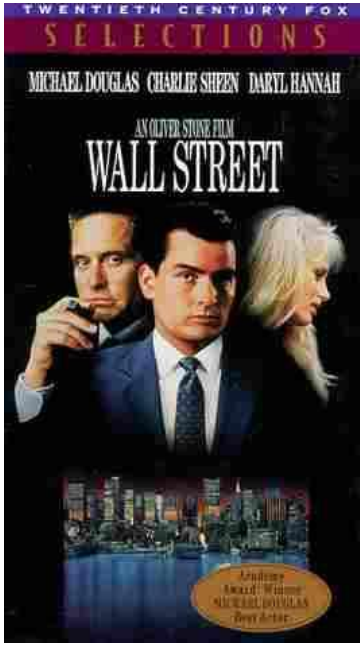
$$\underbrace{c(P) \geq c(P') + c(x,y)}_{\text{nonnegative weights}} \geq_{\text{inductive hypothesis}} d(x) + c(x,y) \geq_{\text{defn of } \pi(y)} \pi(y) \geq_{\text{Dijkstra chose } v \text{ instead of } y} \pi(v)$$

Extra Slides

Exercices: Coin Changing

Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.

- *Gordon Gecko (Michael Douglas)*



Coin Changing

Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

Ex: 34¢.

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Ex: \$2.89.

Coin-Changing: Greedy Algorithm

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

```
Sort coins denominations by value: c1 < c2 < … < cn.

  coins selected
S ← φ
while (x ≠ 0) {
   let k be largest integer such that ck ≤ x
   if (k = 0)
      return "no solution found"
   x ← x - ck
   S ← S ∪ {k}
}
return S
```

Q. Is cashier's algorithm optimal?

Coin-Changing: Analysis of Greedy Algorithm

Theorem. Greed is optimal for U.S. coinage: 1, 5, 10, 25, 100.

Pf. (by induction on x)

- Consider optimal way to change $c_k \le x < c_{k+1}$: greedy takes coin k.
- We claim that any optimal solution must also take coin k.
  - if not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to x
  - table below indicates no optimal solution can do this
- Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by greedy algorithm. ▪

k	c_k	All optimal solutions must satisfy	Max value of coins 1, 2, …, k-1 in any OPT
1	1	$P \le 4$	-
2	5	$N \le 1$	4
3	10	$N + D \le 2$	$4 + 5 = 9$
4	25	$Q \le 3$	$20 + 4 = 24$
5	100	no limit	$75 + 24 = 99$

Coin-Changing: Analysis of Greedy Algorithm

Observation. Greedy algorithm is sub-optimal for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

Counterexample. 140¢.

- Greedy: 100, 34, 1, 1, 1, 1, 1, 1.
- Optimal: 70, 70.

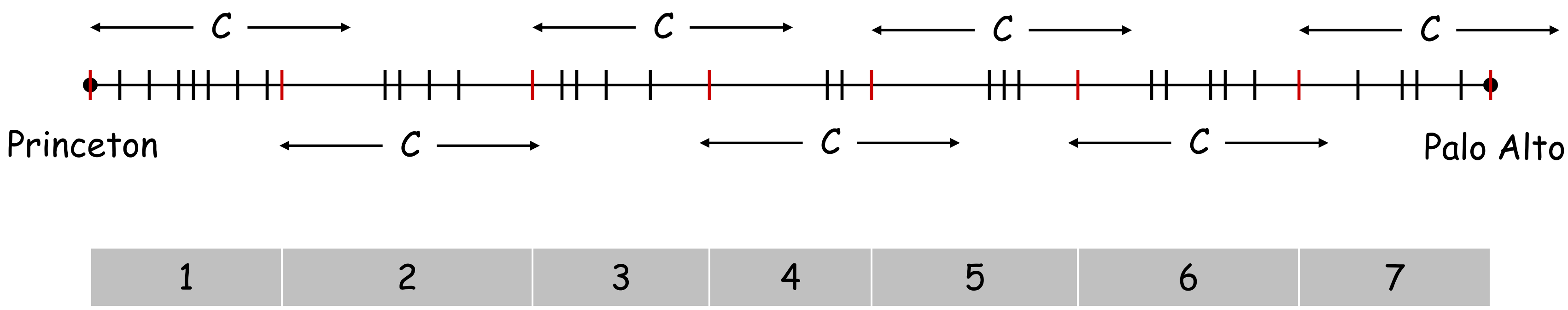
Selecting Breakpoints

Selecting Breakpoints

Selecting breakpoints.

- Road trip from Princeton to Palo Alto along fixed route.
- Refueling stations at certain points along the way.
- Fuel capacity = C.
- Goal: makes as few refueling stops as possible.

Greedy algorithm. Go as far as you can before refueling.



Selecting Breakpoints: Greedy Algorithm

Truck driver's algorithm.

```
Sort breakpoints so that: 0 = b₀ < b₁ < b₂ < ... < bₙ = L

S ← {0}   ← breakpoints selected
x ← 0     ← current location

while (x ≠ bₙ)
   let p be largest integer such that bₚ ≤ x + C
   if (bₚ = x)
      return "no solution"
   x ← bₚ
   S ← S ∪ {p}
return S
```

Implementation. $O(n \log n)$

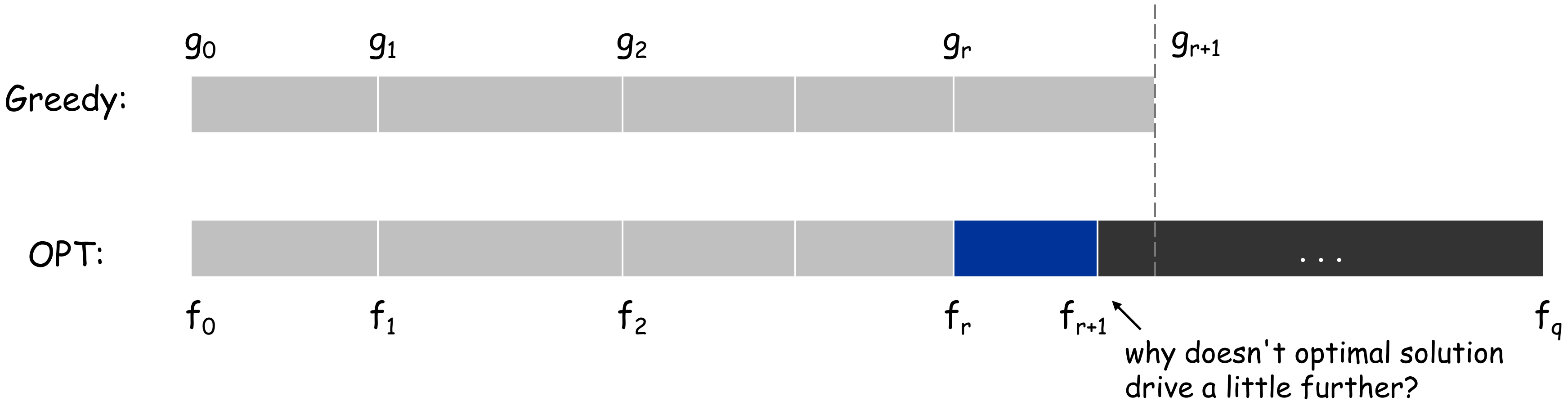
- Use binary search to select each breakpoint p.

Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy.
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in an optimal solution with $f_0 = g_0$, $f_1 = g_1$, . . . , $f_r = g_r$ for largest possible value of r.
- Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm.



Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy.
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in an optimal solution with $f_0 = g_0$, $f_1 = g_1$, . . . , $f_r = g_r$ for largest possible value of r.
- Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm.